

# Game Models for Open Systems<sup>\*</sup>

Luca de Alfaro

Department of Computer Engineering, UC Santa Cruz, USA

**Abstract.** An *open system* is a system whose behavior is jointly determined by its internal structure, and by the input it receives from the environment. To solve control and verification problems, open systems have often been modeled as games between the system and the environment; we argue that the game view of open systems should be extended also to the definitions of system refinement and composition.

We give a symmetrical interpretation to games between system and environment: the moves of the system represent the outputs that the system can generate (the output guarantees), and symmetrically, the moves of the environment represent the inputs that the system can accept (the input assumptions). We argue in favor of defining refinement of open systems in terms of *alternating simulation*, which is the relation between games that plays the same role of simulation between transition systems. Alternating simulation captures the principle that a component refines another if it has weaker input assumptions, and stronger output guarantees. Furthermore, we argue in favor of a notion of composition that accounts for the *compatibility* between input assumptions and output guarantees, and that enables the synthesis of new input guarantees for the composed system. These game-theoretical notions of refinement and compatibility are related to the type-theoretical notions of *subtyping* and *type compatibility*, and give rise to an expressive modeling framework for component-based design and verification.

## 1 Introduction

A basic distinction in concurrency theory is that between *closed* and *open systems*. The behavior of a closed system is completely determined by its internal structure, and it cannot be influenced by the environment. In contrast, the behavior of open systems is jointly determined by their internal structure, and by the inputs received from the environment. Open models can be used to analyze systems that maintain an ongoing interaction with their environment (or *reactive systems* [27, 28]), such as embedded systems and control systems. Moreover, open models can be used to study the individual components of larger designs, whose behavior usually depends on the inputs they receive from other components.

Closed systems are naturally modeled in terms of *transition systems* [23]. A transition system consists in a set of system states, and in a set of transitions among the states. Whenever multiple transitions are available from a state,

---

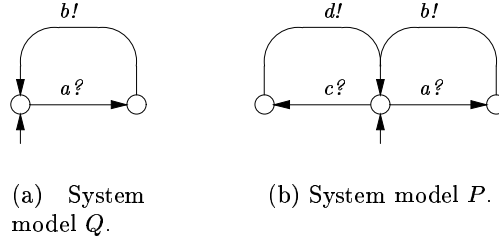
<sup>\*</sup> This research was supported in part by the NSF CAREER award CCR-0132780, the NSF grant CCR-0234690, and the ONR grant N00014-02-1-0671.

Report Documentation Page				Form Approved OMB No. 0704-0188	
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE <b>2006</b>		2. REPORT TYPE		3. DATES COVERED <b>00-00-2006 to 00-00-2006</b>	
4. TITLE AND SUBTITLE <b>Game Models for Open Systems</b>				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) <b>University of California, Department of Computer Engineering, Santa Cruz, CA, 95064</b>				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT <b>Approved for public release; distribution unlimited</b>					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES <b>21</b>	19a. NAME OF RESPONSIBLE PERSON
a. REPORT <b>unclassified</b>	b. ABSTRACT <b>unclassified</b>	c. THIS PAGE <b>unclassified</b>			

one of them is selected nondeterministically: this nondeterminism can be used to represent uncertainty, or freedom of implementation, in the system model. Games generalize transition systems by providing a model for multiple independent sources of nondeterminism. Each source of nondeterminism is represented as a player, whose possible moves correspond to the nondeterministic choices available to the source. In particular, two-player games have proven to be an expressive model for open systems, enabling the distinction between the nondeterministic choices that originate within the system (such as the choice among possible outputs), and the choices that originate in its environment (such as the choice of inputs to the system). Game models have been widely used to analyze and solve control problems for open systems [1, 32, 33]. The game view has also been used in the specification and verification of the interaction between components, and of the interaction between components and their environment [24, 16, 19].

We argue that games constitute a natural model for open systems, and we argue that not only control and verification, but also *refinement* and *composition* of open systems should be phrased in game-theoretic terms [14, 15]. In other words, we argue in favor of adopting games as the mathematical structure underlying open system models, and in favor of phrasing all notions related to open system — from composition, to refinement, to specification and verification — in terms of games between two players: Input (representing the environment) and Output (representing the system). The two players specify the behavior of the system: the moves of Input correspond to inputs it can receive from the environment, the moves of Output correspond to the outputs it can produce. Using games, rather than transition systems, in the definitions of refinement and composition, enables to keep distinct, and treat differently, the roles of inputs and outputs. In composition, game models can account for the causal relationship between outputs and inputs, and can distinguish between the situation in which a component produces fewer outputs than another can accept as inputs — which indicates compatibility — and the situation in which a component generates more outputs than another can accept as input — which indicates incompatibility. Refinement of transition systems is defined as behavior containment: roughly, all nondeterministic choices of the implementation must be possible (and thus permitted) also in the specification. In games, on the other hand, the natural notion of refinement is *alternating refinement* [8]: when the two players are Input and Output, alternating refinement holds when the implementation can accept more input behaviors, and produce fewer output behaviors, than the specification. Thus, games provide a notion of *compatibility* between system components, and a notion of *refinement* that preserves this compatibility. This leads to a uniform framework for the study of control, verification, component-based design, and implementation of open systems.

This paper is divided in two parts. In the first part, consisting of Sections 2, 3, and 4, we outline our informal arguments in favor of the adoption of game models for open systems. In the second part, consisting of Sections 5, 6, and 7, we



**Fig. 1.** Refinement of game models. We denote input actions with question marks, and output actions with exclamation marks.

illustrate the approach by presenting a simple, concrete game model for untimed asynchronous systems, derived from [14].

## 2 Overview: Refinement

The notion of refinement aims at capturing the relation between an abstract model of a component, and a more detailed model, or between a model expressing a specification, and a model describing an implementation. The definitions of transition-system refinement equate refinement to behavior containment: all the behaviors of the implementation (or of the detailed model) must be also behaviors of the specification (or of the abstract model). In particular, refinement is commonly defined as trace inclusion, or simulation [30]. This definition is well-suited to closed systems, which are not influenced by their environment: roughly, the definition ensures that if all the behaviors of the specification model are “correct”, so are all behaviors of the implementation. In particular, if the specification satisfies a property expressed in linear-time temporal logic [27] or ACTL [12], so does the implementation.

For open systems, however, behavior containment is a somewhat unsatisfactory definition of refinement, as it requires also that all the *input* behaviors of the implementation are a subset of those of the specification. Informally, this would mean that the implementation would be able to accept *fewer* input behaviors than the specification. As an example, consider the system models depicted in Figure 1. The model  $Q$  represents a system that can accept an input  $a$ , to which it replies with output  $b$ . The model  $P$  represents a system that can accept both input  $a$ , to which it replies again with output  $b$ , and also input  $c$ , to which it replies with output  $d$ . Clearly, if we need a component with the functionality of  $Q$ , and we are given  $P$  instead, we can just use  $P$ , disregarding its additional input. In other words,  $P$  should count as a correct implementation of  $Q$ . However, neither  $Q$  simulates  $P$ , nor is the language of  $P$  a subset of the language of  $Q$ : indeed, exactly the opposite holds.

Defining refinement of open systems as behavior containment is also at variance with the notion of subtyping in type theory: there, a functional type

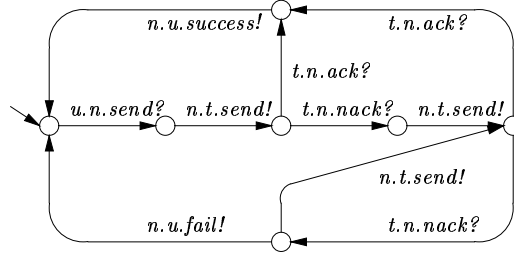
$\tau : \sigma_i \rightarrow \sigma_o$  is a *subtype* of  $\tau' : \sigma'_i \rightarrow \sigma'_o$  (written  $\tau \preceq \tau'$ ) if  $\tau$  accepts more inputs ( $\sigma'_i \preceq \sigma_i$ ) and produces fewer outputs ( $\sigma_o \preceq \sigma'_o$ ) [31]. Subtyping is thus *contravariant* with respect to inputs and outputs, in contrast with simulation, which is *covariant*. This suggests to replace simulation, or trace containment, with a notion that is contravariant with respect to inputs and outputs. Similarly to subtyping, the notion should encode the intuition that when  $P$  refines  $Q$ , then:

1.  $P$  accepts at least as many input behaviors as  $Q$  does, so that we can use  $P$  whenever we used  $Q$  without causing an (input) incompatibility;
2. moreover, when  $P$  and  $Q$  are subjected to the same input behavior,  $P$  should produce a subset of the output behaviors of  $Q$ .

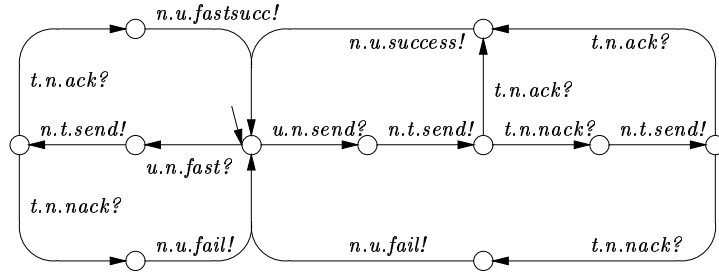
If we model each of  $P$  and  $Q$  as a two-player game between a player *Input* (responsible of the input nondeterminism) and a player *Output* (responsible of the output nondeterminism), then these requirements are captured by the notions of game refinement proposed in [8], namely, *alternating trace containment* and *alternating simulation*. Alternating simulation is a relation between the states of  $P$  and the states of  $Q$  such that, at related states, all the outputs that can be generated by  $P$  can also be generated by  $Q$ , and all the inputs that can be accepted by  $P$  can be accepted by  $Q$ ; moreover, corresponding inputs and outputs lead to states of  $P$  and  $Q$  that are again related. In this paper, we adopt alternating simulation as the notion of refinement between open systems.

Figure 2 illustrates two game models that are related by alternating simulation. Model *Netw* represents a very simple component of a network stack ( $n$ ), that interacts with a user level ( $u$ ) and a transport layer ( $t$ ). An action of the form  $n.u.a$  indicates that an action  $a$  is sent from  $n$  to  $u$  (and is therefore an output of  $n$ ). The model represents a component that, upon receiving a command to send a packet (input action  $u.n.send$ ), invokes the transport layer (output action  $n.t.send$ ). If the transport layer succeeds in sending the packet, it informs the network layer (input  $t.n.ack$ ), and the network layer returns success (output  $n.u.success$ ) to the upper layer. Otherwise, if the transport layer fails (input  $t.n.nack$ ), the network layer invokes the transport layer at least one more time, and then reports success or failure (output  $n.u.fail$ ) to the upper layer. The model *NcompPlus* is similar, except that if the transport layer fails during a request  $n.u.send$ , the network layer tries exactly once more to send the packet, and reports then failure or success. Moreover, in addition to  $u.n.send$ , *NcompPlus* accepts also another input  $u.n.fast$ , that specifies that only one attempt should be made at delivering the packet. Success of this additional service is reported by the output  $n.u.fastsucc$ .

Intuitively, the module *NcompPlus* is a proper implementation of *Netw*. Under the same inputs, *NcompPlus* produces a subset of the behaviors (retrying transmission exactly once, rather than at least once); *NcompPlus* also can perform an additional service, which does not prevent us from using *NcompPlus* in place of *Netw*. Indeed, the corresponding states of *Netw* and *NcompPlus* are related by alternating simulation. On the other hand, there is obviously no simulation relation (in either direction) between *Netw* and *NcompPlus*.



(a) *Netw*



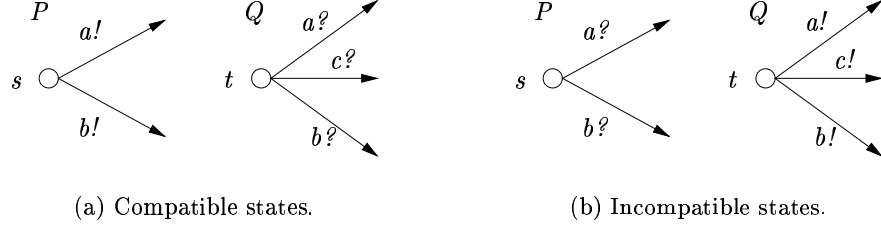
(b) *NcompPlus*

**Fig. 2.** Refinement of game models.

### 3 Overview: Composition

In transition systems, the presence of a single notion of nondeterminism is reflected in the definition of composition, where input and output transitions are treated in the same fashion. In the process algebras CCS [29], CSP [22] and ACP [9], composition is defined as conjunction: a shared action can occur in a composite system only if it can occur in all the components. Thus, the choice of which output to produce, and the choice of which input to accept, are governed by a single source nondeterminism, and two processes, when composed, “cooperate” to effect a transition that is possible for both. Composition as conjunction leads to a flexible modeling paradigm, in which composition can be used both to add new components to a system, and to constrain already present components to a subset of their behaviors [22].

Composition as conjunction, however, does not capture the cause-effect relationship between outputs and inputs. For instance, this approach does not distinguish between the situation of Figure 3(a), where a system  $P$  that produces some outputs is composed with a system  $Q$  that can accept a superset of those inputs, and the situation of Figure 3(b), where a system  $Q$  that produces



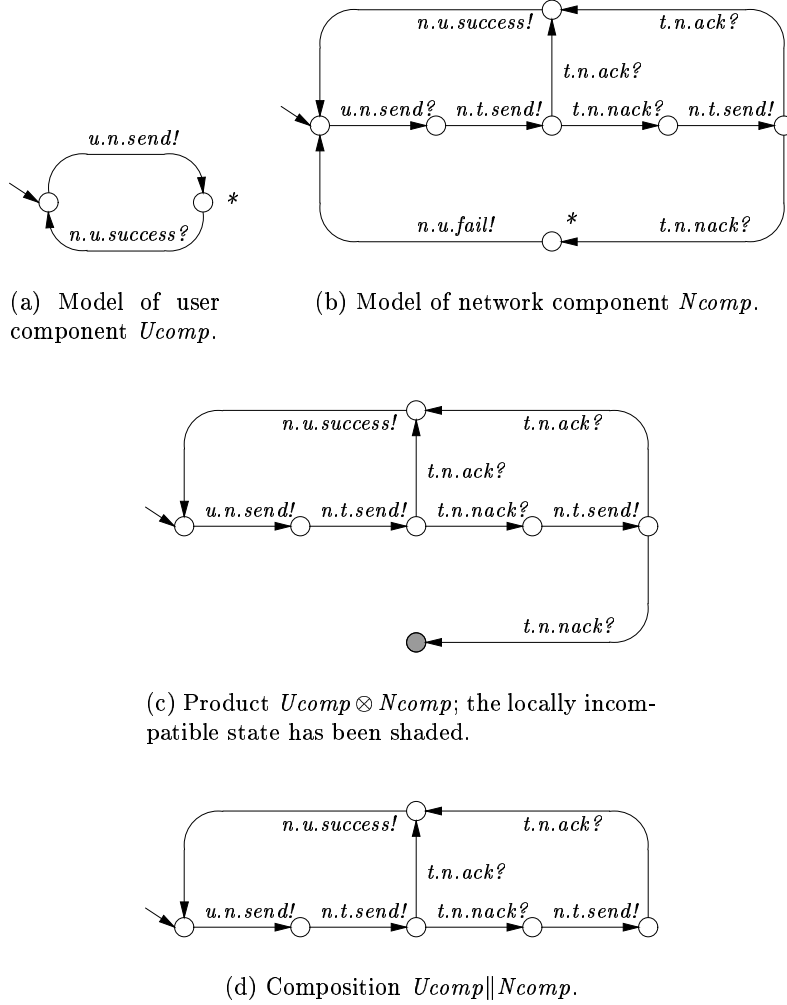
**Fig. 3.** Causality and compatibility.

some outputs is composed with a system  $P$  that can accept only a subset of those outputs. In both cases, only the shared action  $a$  and  $b$  can occur. In Figure 3(a), this agrees with the notion of causality: since the output  $c!$  cannot occur, the action  $c$  does not occur in the composition. In Figure 3(b), however, the fact that input  $c?$  cannot be accepted by  $P$  causes  $Q$  never to emit  $c!$ , reversing the intuitive causal dependency of inputs on outputs.

To restore the causal dependency of inputs on outputs, several modeling languages, such as I/O Automata [25], SMV [13], and Reactive Modules [7] stipulate that systems must be able to accept all possible inputs: this is the so-called *input enabled* approach to composition. While this approach restores input/output causality, it does so at the cost of removing the ability to express which inputs are allowed: input-enabled models constrain (and thus model) output behavior only.

Game models, such as interface theories [14, 15] restore causality by stipulating that output choice takes the precedence over input choice, and by introducing a notion of compatibility. The choice of outputs is independent from the set of acceptable inputs: if a system chooses an output that cannot be accepted as input by another system, an *incompatibility* occurs. Precisely, a state  $s$  of a system  $P$  is compatible with a state  $t$  of a system  $Q$  if (i) all outputs that can be generated by  $P$  at  $s$ , and that are in the input language of  $Q$ , can be accepted at  $t$ , and symmetrically, (ii) all outputs that can be generated by  $Q$  at  $t$ , and that are in the input language of  $P$ , can be accepted at  $s$ . For instance, the two states of Figure 3(a) are compatible, while the two states of Figure 3(b) are not. Such incompatibilities between states are, however, a *local* notion: even if two component models  $P$  and  $Q$  have states that are pairwise incompatible, they may still be compatible with one another. In fact, each of  $P$  and  $Q$  imposes some requirements on the environment, through the specification of the inputs it can accept. Hence, when composing  $P$  and  $Q$ , the interesting question is not whether  $P$  and  $Q$  work together correctly in all environments (not even  $P$  and  $Q$  in isolation do so), but rather, whether there is *some* environment in which they can work correctly together. This question can be cast, and answered, in game-theoretic terms by considering again a game between the Input and Output players. The models  $P$  and  $Q$  are *compatible* if Input has a winning strategy

that ensures that local incompatibilities never arise; all such winning strategies form in turn the allowed input behaviors for  $P\|Q$ . Thus, the input behaviors allowed by  $P\|Q$  correspond exactly to the environment behaviors that ensure that  $P$  and  $Q$  work together correctly.



**Fig. 4.** Product and composition of game models.

This approach to compatibility and composition is illustrated by the following example [14]. Consider the model  $Ncomp$  (Figure 4(b)), representing again a network layer that, when asked to send a packet, uses an underlying transport

packet that makes at most two attempts at sending the packet. Assume that this network model is composed with an upper layer  $Ucomp$  (Figure 4(a)) that tries to send a packet, and expects a successful result. In the composition, the two states denoted by (\*) will be incompatible: in fact, at one state component  $Ncomp$  generates the output  $n.u.fail$ , which cannot be accepted as input at the other state. Hence, in the automata product of  $Ncomp$  and  $Ucomp$  (Figure 4(c)), the corresponding state is locally incompatible. Nevertheless, the Input player has a strategy to avoid entering the incompatible state, that consists in never providing the  $t.n.ack$  transition leading to it. The composition  $Ncomp \parallel Ucomp$ , depicted in Figure 4(d), consists of the states and edges that can be traversed when Input avoids all local incompatibilities.

The proposed notions of compatibility and refinement preserve *substitutivity of refinement*: if a component model  $P$  refines  $Q$ , or  $P \preceq Q$ , and if  $Q$  is compatible with a component model  $R$ , then also  $P$  is compatible with  $R$ .

We note that, as was the case for refinement, also the above notion of compatibility (and composition) for game models has close parallels in type theory. When a type  $\tau : \rho \rightarrow \sigma$  is composed with  $\tau' : \rho' \rightarrow \sigma'$  to form  $\tau' \circ \tau$ , the (functional) composition  $\tau' \circ \tau$  satisfies type compatibility if  $\sigma \preceq \rho'$ , that is, if  $\sigma$  is a subtype of  $\rho'$ . If  $\sigma$  and  $\rho'$  are simple sets of values, this requirement amounts to  $\sigma \subseteq \rho'$ : type compatibility means that the possible outputs  $\sigma$  are a subset of the accepted inputs  $\rho'$ . With game models such as game theories, however, we go one step beyond, exploiting the fact that we have a representation of  $\tau$  as a function  $\tau : \rho \mapsto \sigma$ , rather than as a type  $\rho \rightarrow \sigma$ . When  $\sigma \not\subseteq \rho'$ , we can thus generate the weakest (the largest, in terms of sets)  $\tilde{\rho}$  such that  $\tilde{\rho} \subseteq \rho$ , and such that  $\tau$ , when restricted to the domain  $\tilde{\rho}$ , has an image that is a subset of  $\rho'$  (that is,  $\tau(\tilde{\rho}) \subseteq \rho'$ ).

## 4 Discussion

### 4.1 Open Systems as Games

In summary, we argue in this paper for an approach to the modeling of open systems guided by the following principles:

1. **Distinguish Input and Output nondeterminism.** Input and output nondeterminism are kept separate, and they are modeled as the actions of two players, *Input* and *Output*.
2. **Refinement as alternating simulation.** A system model  $P$  refines a system model  $Q$  if there is a relation between their state spaces such that, at related states, every move of Input of  $Q$  can be performed in  $P$ , and every move of Output in  $Q$  can also be done in  $P$ ; moreover, the transition caused by corresponding moves leads again to related states.
3. **Composition as Input/Output game.** The composition of two components  $P$  and  $Q$  is defined in terms of two notions: a notion of *local compatibility*, and a game between the Input and Output players. Local compatibility captures causality: when a shared output is chosen, it should also be accepted

as an input. The game between Input and Output is used to synthesize global input assumptions that guarantee local compatibility. If these input assumptions are satisfiable, then  $P$  and  $Q$  are said to be *compatible*, and the input assumptions are used to constrain  $P||Q$ .

There are many models of computation: for instance, composition can be synchronous or asynchronous, and the systems can be untimed, or real-time. Correspondingly, there is no single notion of game model, but rather, game models have been proposed for each of the settings. These models have been called *interface theories*, due to their ability to capture the input/output interaction among components; interface theories have been presented for untimed asynchronous [14] and synchronous [15, 10] models, as well as for real-time models [17] and for modeling resource requirements [11]. In the remainder of the paper, we illustrate in more detail the game approach to the modeling of open systems by presenting in detail *interface automata*, an asynchronous untimed model that is very similar to the one presented in [14].

## 4.2 Related Work

Most, if not all, of the individual ideas on which game models are based had been introduced earlier in the literature. The idea of specifying independent constraints for input and output behavior is present in *trace theories* [18], and indeed in the untimed, asynchronous case our game composition is essentially the composition of prefix-closed trace structures, followed by a normal-form reduction. The notion of *conformation* in trace theories is also closely related to our notion of refinement:  $P$  conforms to  $Q$  if  $P$  accepts more inputs (has fewer failure traces) and produces fewer outputs (has fewer successful traces) than  $Q$ . Alternating simulation (and alternating trace containment) was introduced later, in [8], and trace theories do not formulate conformation in terms of alternating simulation. The distinction between internal and external choice is present in many modeling languages, an early example being CSP [22]. All the algorithms used in untimed game models are standard. Refinement is based on alternating simulation [8]. Composition involves a compatibility check that can be solved as an instance of controller synthesis [1, 32–34].

Thus, the novelty in the outlined game-based approach to open systems systems is not in any isolated algorithmic aspect, but rather in the recognition that refinement, composition, verification, and synthesis can be homogeneously cast as game relations and problems. In many respects, the work on game semantics for programming languages [3, 5, 6] and process algebras [2, 4] achieves a similar realization that games provide a unified model for interaction, and indeed game semantics are very close in spirit to our interface models. The main difference is that in game semantics the definitions take *strategies* as the central object, while the emphasis in interface theories has been on automata-like models. Moreover, the difference in application areas (programming languages vs. untimed, or real-time systems) motivates differences in the mathematical setup that mask the underlying similarity of the two approaches.

### 4.3 On Modeling Open Systems as Transition Systems

In light of the differences between the transition-system and game models of open systems, one may ask why the limitations of the transition-system approach have not been a greater drawback in practice. Broadly, the answer is that input-enabled formalisms, while preventing the specification of input constraints, are adequate for answering many verification questions, and verification, rather than design, has been the main application of formal methods.

Specifically, the chief advantage of the game-based approach in composition is the ability to define input constraints, and the availability of a notion of compatibility. These notions are helpful in the component-based design of systems, where it is important to encode not only a component's behavior, but also its input assumptions, in order to replace or refine components while respecting the design assumptions of the other components. These notions are however less central when the goal is to construct a model of an existing system and to verify, once all components have been composed, that it satisfies a specification.

With respect to refinement, we note first that in many cases the covariant definition of refinement (trace inclusion or simulation) was often applied to *input-enabled systems*, such as I/O Automata [25] and Reactive Modules [7]. For these models, since there are no constraints on input behavior, the requirements about refinement of input behavior are vacuous. Hence, a definition that essentially refers to output behavior only is satisfactory. The inability to encode directly input assumptions, however, leads to a pitfall in the straightforward application of refinement checking. Often, a refinement  $P \preceq Q$  between an input-enabled implementation  $P$  and a specification  $Q$  does not hold, because we ask  $P$  to exhibit output behaviors allowed by  $Q$  under *all* inputs, including possibly inputs that are known not to occur in the intended usage of  $P$ , and under which the behavior of  $P$  was not given any consideration during design. Usually, this situation is described by saying that more *environment assumptions* are needed in order to prove  $P \preceq Q$ . Models that are not required to be input enabled can capture input assumptions directly. In an input-enabled model, however, these input assumptions can only be captured by writing a separate model  $E$  for the environment, and by composing this model with  $P$ : the goal becomes thus to prove  $E \parallel P \preceq Q$ . This approach has been very effective in practice, and it has been extended to enable the compositional verification of refinement relations [21, 20]. However, we observe that  $E \parallel P$  is now a closed system, since all inputs of  $P$  are provided by  $E$ : hence, in this approach refinement is checked essentially between a *closed* implementation and a specification. Thus, the success in applying a covariant notion of refinement to open systems can be explained by the fact that the notion has been mostly applied either to systems with trivial input behavior (such as input-enabled systems), or to closed systems (such as systems composed with their environment assumptions).

## 5 Interface Automata

To illustrate concretely the ideas of the previous sections, we now present in detail *interface automata*, a game model for asynchronous systems derived from [14]. Essentially, an interface automaton is a deterministic labeled transition system in which the labels correspond to input and output actions; the “game” aspect is only evident in the definitions of refinement and composition, given in the next sections. The formal definition is as follows.

**Definition 1 (interface automaton).** An interface automaton  $P = \langle S_P, s_P^{init}, \mathcal{A}_P^I, \mathcal{A}_P^O, \Gamma_P^I, \Gamma_P^O, \delta_P \rangle$  consists of the following components:

- $S_P$  is a set of *states*.
- $s_P^{init} \subseteq S_P$  is the set of *initial states*. We require that  $s_P^{init}$  contains at most one state. If  $s_P^{init} = \emptyset$ , then  $P$  is called *empty*.
- $\mathcal{A}_P^I$  and  $\mathcal{A}_P^O$  are mutually disjoint sets of *input* and *output* actions. We denote by  $\mathcal{A}_P = \mathcal{A}_P^I \cup \mathcal{A}_P^O$  the set of all *actions*.
- $\Gamma_P^I : S_P \mapsto 2^{\mathcal{A}_P^I}$  assigns to each state  $s \in S_P$  a (possibly empty) set of *input moves*, and  $\Gamma_P^O : S_P \mapsto 2^{\mathcal{A}_P^O}$  assigns to each state  $s \in S_P$  a (possibly empty) set of *output moves*. The input moves represent the actions that can be accepted at  $s$ , and the output moves represent the actions that can be generated at  $s$ . For  $s \in S_P$ , we denote by  $\Gamma_P(s) = \Gamma_P^I(s) \cup \Gamma_P^O(s)$  the set of all actions at  $s$ .
- $\delta_P : S_P \times (\mathcal{A}_P^I \cup \mathcal{A}_P^O) \mapsto S_P$  is a *transition function* that associates with each state  $s \in S_P$  and action  $a \in \mathcal{A}_P$  a *destination* state  $\delta_P(s, a) \in S_P$ . ■

In the following, we refer to the components of an interface automaton  $P$  by  $S_P$ ,  $s_P^{init}$ ,  $\mathcal{A}_P^I$ ,  $\mathcal{A}_P^O$ ,  $\Gamma_P^I$ ,  $\Gamma_P^O$ , and  $\delta_P$ . The set  $\mathcal{A}_P^I \setminus \Gamma_P^I(s)$  of *illegal inputs* at a state  $s \in S_P$  consists of the input actions of  $P$  that cannot be accepted at  $s$ . Non-empty interface automata have a single initial state; empty interface automata arise when incompatible automata are composed, as we will see in Section 7. We say that  $P$  is *closed* if  $\mathcal{A}_P^I = \emptyset$ ; otherwise, we say that  $P$  is *open*.

A *strategy* restricts the set of input or output moves that can be played; as is common for games, strategies (and thus the restrictions) can in general depend on the past.

**Definition 2 (strategy).** An *input* (resp. *output*) strategy for  $P$  is a mapping  $\pi^I : S_P^+ \mapsto 2^{\mathcal{A}_P^I}$  (resp., a mapping  $\pi^O : S_P^+ \mapsto 2^{\mathcal{A}_P^O}$ ) such that, for all  $s \in S_P$  and all  $\sigma \in S_P^*$ , we have  $\pi^I(\sigma s) \subseteq \Gamma_P^I(s)$  (resp.  $\pi^O(\sigma s) \subseteq \Gamma_P^O(s)$ ). We denote by  $\Pi_P^I$  and  $\Pi_P^O$  the set of input and output strategies of  $P$ , respectively. ■

An input and an output strategy jointly determine a *set* of traces in  $S_P^+$ : at each step, if the input strategy proposes a set  $\mathcal{B}^I$  of actions, and the output strategy proposes a set  $\mathcal{B}^O$ , an action from  $\mathcal{B}^I \cup \mathcal{B}^O$  is selected nondeterministically. Since our definitions of compatibility and composition do not require the consideration of progress properties, we define the outcomes of strategies in terms of finite traces.

**Definition 3 (outcomes).** Given a state  $s \in S_P$ , an input strategy  $\pi^I \in \Pi_P^I$  and an output strategy  $\pi^O \in \Pi_P^O$ , the set  $\text{Outcomes}_P(s, \pi^I, \pi^O) \subseteq S_P^+$  is the smallest set defined inductively by the following clauses:

- $s \in \text{Outcomes}_P(s, \pi^I, \pi^O)$ ;
- if  $\sigma t \in \text{Outcomes}_P(s, \pi^I, \pi^O)$  for  $\sigma \in S_P^+$  and  $t \in S_P$ , then for all  $a \in \pi^I(\sigma t) \cup \pi^O(\sigma t)$  we have  $\sigma t \delta_P(s, a) \in \text{Outcomes}_P(s, \pi^I, \pi^O)$ . ■

We say that a state  $s \in S_P$  *occurs* in a trace  $\sigma \in S_P^+$ , written (by abuse of notation)  $s \in \sigma$ , if  $\sigma = s_0 s_1 s_2 \cdots s_k$  with  $s_i = s$  for some  $0 \leq i \leq k$ . A state  $s \in S_P$  is *reachable* in  $P$  if there is a sequence of states  $s_0, s_1, \dots, s_n$  with  $s_0 \in s_P^{\text{init}}$ ,  $s_n = s$ , and such that for all  $0 \leq k < n$  there is  $a_k \in \Gamma_P(s_k)$  such that  $\delta_P(s_k, a_k) = s_{k+1}$ .

*Action naming.* A useful convention for action names consists in assuming a fixed set  $\mathcal{U}$  of *components*, and a fixed set  $\mathcal{N}$  of *signal names* (these sets need not be finite). Then, an action consists in a triple  $\langle U, V, a \rangle$ , where  $U \in \mathcal{U}$  is the component that can generate the action as output,  $V \in \mathcal{U}$  is the component that can accept the action as input, and  $a \in \mathcal{N}$  is a name used to distinguish this particular action from all other actions that correspond to communication from  $U$  to  $V$ . An interface automaton models a set of components. If this convention is adopted, then an interface automaton  $P$  that models a set of components  $\mathcal{V} \subseteq \mathcal{U}$  has set of input actions  $\mathcal{A}_P^I = (\mathcal{U} \setminus \mathcal{V}) \times \mathcal{V} \times \mathcal{N}$  and set of output actions  $\mathcal{A}_P^O = \mathcal{V} \times (\mathcal{U} \setminus \mathcal{V}) \times \mathcal{N}$ . This convention enables the drawing of interface automata omitting the explicit mention of their sets of input and output actions. The automata in Figures 2 and 4 have been drawn using this convention, using a dot “.” to separate the components of the triples, and using  $n$  and  $u$  as the names for the network and user components.

## Discussion

An interface automaton represents in a joint fashion both the input behavior of a component (which input sequences can be accepted), and the output behavior of the component (which output sequences can be generated). One may wonder whether the same information could be represented by two separate models, one describing only the input behavior, the other only the output behavior, and whether it would be possible to use already established formalisms to encode these two models.

The answer is, trivially, affirmative. It is easy to see that all the information contained in an interface automaton  $P$  could be encoded by a pair of I/O automata  $P_I$  and  $P_O$ , where the automaton  $P_O$  is input-enabled, and describes only the output behavior, and the automaton  $P_I$  is “output-universal” (the symmetrical notion of input-enabled), and describes only input behavior. Precisely, we let  $\Gamma_{P_O}^I(s) = \mathcal{A}_P^I$ , and symmetrically,  $\Gamma_{P_I}^O(s) = \mathcal{A}_P^O$  for all  $s \in S_P$ ; the transition function  $\delta_P$  is extended arbitrarily for these new actions, yielding  $\delta_{P_O}$  and  $\delta_{P_I}$ . All other components of  $P_O$  and  $P_I$  coincide with their corresponding component

in  $P$ . Then, it is immediate to see that all information in  $P$  can be reconstructed by considering the synchronous composition of  $P_I$  and  $P_O$ . It is also clear that  $P_O$  is an I/O automaton [26], and  $P_I$  is essentially an I/O automaton, except that the roles of inputs and outputs are exchanged.

Indeed, the difference between game models, such as interface automata, and transition system models, such as I/O automata, does not lie in the particular syntax chosen for representing a model, but rather, in how the operations on the models, such as refinement and composition, are defined.

## 6 Refinement

Refinement of interface automata is defined in terms of *alternating simulation*, which is the extension of the classical notion of simulation to games [8]. Informally, an alternating simulation  $\rho \subseteq S_P \times S_Q$  from  $P$  to  $Q$  is a relation such that, whenever  $(s, t) \in \rho$ , then all input moves of  $s$  can be simulated by  $t$ , and conversely, all output moves of  $t$  can be simulated by  $s$ . The definition is as follows.

**Definition 4 (alternating simulation).** An *alternating simulation relation*  $\rho$  from  $P$  to  $Q$  is a relation  $\rho \subseteq S_P \times S_Q$  such that, for all  $\langle s, t \rangle \in \rho$  and all  $a \in \Gamma_Q^I(t) \cup \Gamma_P^O(s)$  we have:

$$\Gamma_Q^I(t) \subseteq \Gamma_P^I(s) \quad \Gamma_P^O(s) \subseteq \Gamma_Q^O(t) \quad \langle \delta_P(s, a), \delta_P(t, a) \rangle \in \rho \quad \blacksquare$$

Refinement is defined as the existence of an alternating simulation between initial states.

**Definition 5 (refinement).** An interface automaton  $P$  *refines* an interface automaton  $Q$ , written  $P \preceq Q$ , if the following conditions hold:

1.  $\mathcal{A}_P^I \subseteq \mathcal{A}_Q^I$ ;
2.  $\mathcal{A}_P^O \subseteq \mathcal{A}_Q^O$ ;
3. there is an alternating refinement relation  $\rho$  from  $P$  to  $Q$ , a state  $s \in s_P^{init}$ , and a state  $u \in s_Q^{init}$  such that  $\langle s, u \rangle \in \rho$ .  $\blacksquare$

The third condition states that there is an alternating simulation relation from  $P$  to  $Q$  that relates the initial state of  $P$  to that of  $Q$ . The first condition, together with the third, ensures that if two states  $s$  and  $t$  are related by an alternating simulation, then the illegal inputs at  $s$  are a subset of those at  $t$ . The third condition ensures that  $P$  does not have any extra output, compared to  $Q$ , that could clash with outputs of other components in a design. As we will see in Section 7, these conditions ensure that if  $Q$  is compatible with  $R$ , and  $P \preceq Q$ , then  $P$  is compatible with  $R$ .

Conditions 1 and 2 state that  $Q$  must establish a reserved name space,  $\mathcal{A}_Q^I$  and  $\mathcal{A}_Q^O$ , through which the implementations of  $Q$  are constrained to communicate with the rest of the system. We remark that if the action naming convention of

Section 5 is followed, and if  $P$  and  $Q$  both model the same set of components, then the conditions 1 and 2 are guaranteed to hold.

In addition to refinement, we can introduce a notion of equivalence based on alternating bisimulation. In our asynchronous setting, alternating bisimulation is defined as follows.

**Definition 6 (alternating bisimulation).** An *alternating bisimulation relation*  $\rho$  between  $P$  and  $Q$  is a relation  $\rho \subseteq S_P \times S_Q$  such that, for all  $(s, t) \in \rho$ , we have:

$$\Gamma_Q^I(t) = \Gamma_P^I(s) \quad \Gamma_P^O(s) = \Gamma_Q^O(t)$$

and for all  $a \in \Gamma_Q^I(t) \cup \Gamma_P^O(s)$ ,

$$(\delta_P(s, a), \delta_Q(t, a)) \in \rho \quad \blacksquare$$

Thus, in our asynchronous setting, alternating bisimulation coincides with usual bisimulation, with the additional requirement that input actions can be related only to input actions, and output actions only to output actions. We say that two interfaces are *bi-equivalent* if they have the same sets of input and output actions, and if their initial states are bisimilar.

**Definition 7 (bi-equivalence).** Two interface automata  $P$  and  $Q$  are *bi-equivalent*, written  $P \simeq Q$ , if the following conditions hold:

1.  $\mathcal{A}_P^I = \mathcal{A}_Q^I$  and  $\mathcal{A}_P^O = \mathcal{A}_Q^O$ ;
2. there is an alternating bisimulation relation  $\rho$  between  $P$  and  $Q$ , a state  $s \in s_P^{init}$ , and a state  $u \in s_Q^{init}$  such that  $(s, u) \in \rho$ .  $\blacksquare$

The following theorem summarizes the main properties of refinement and bi-equivalence.

**Theorem 1 (properties of refinement and bi-equivalence).** *The following assertions hold:*

1. *Refinement is reflexive and transitive.*
2. *Bi-equivalence is an equivalence relation: it is reflexive, symmetrical, and transitive.*
3. *Bi-equivalence implies refinement.*

An immediate corollary of this theorem is that bi-equivalent interface automata can be substituted while preserving refinement.

**Corollary 1.** *The following assertions hold for all interface automata  $P$ ,  $Q$ , and  $R$ :*

1.  $P \simeq Q$  and  $P \preceq R$  implies  $Q \preceq R$ ;
2.  $P \simeq Q$  and  $R \preceq P$  implies  $R \preceq Q$ .

## 7 Composition

Two interface automata are composable if they do not share output actions.

**Definition 8.** Two interface automata  $P$  and  $Q$  are *composable* if  $\mathcal{A}_P^O \cap \mathcal{A}_Q^O = \emptyset$ .

■

We define the composition of interface automata in two stages, first defining the *product* automaton  $P \otimes Q$  of two composable interface automata  $P$  and  $Q$ , and then explaining how to construct the *composition*  $P \parallel Q$  from this product.

### 7.1 Product

In the product automaton  $P \otimes Q$ , the interface automata  $P$  and  $Q$  synchronize their shared actions  $\mathcal{A}_P \cap \mathcal{A}_Q$ , and they interleave asynchronously all other actions. Thus, the product of interface automata is similar to the composition of I/O automata [26], except that since interface automata need not be input enabled, there is no guarantee that a shared output action performed by one interface automaton will be part of the product. We denote by  $\text{Shrd}(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q$  the shared actions of  $P$  and  $Q$ , and we denote by  $\text{Prv}(P, Q) = (\mathcal{A}_P \cup \mathcal{A}_Q) \setminus \text{Shrd}(P, Q)$  all other actions. Among the shared actions, we let the *communication actions* be  $\text{Comm}(P, Q) = (\mathcal{A}_P^O \cap \mathcal{A}_Q^I) \cup (\mathcal{A}_P^I \cap \mathcal{A}_Q^O)$ . The set of states of the product is  $S_{P \otimes Q} = S_P \times S_Q$ . An input move  $a$  is available at a state  $\langle s, t \rangle$  of  $P \otimes Q$  if it is available at  $s$  or  $t$ , and if  $a$  is not a communication action. An output move  $a$  is available at  $\langle s, t \rangle$  if one of the following two conditions hold:

- $a$  is not shared, and the output move  $a$  is available at  $s$  or at  $t$ ;
- $a$  is a communication action, and it occurs at  $s$  as an output move and at  $t$  as an input move, or vice versa, at  $t$  as an output move and at  $s$  as an input move.

The precise definition is as follows.

**Definition 9 (interface automata product).** If  $P$  and  $Q$  are composable interface automata, their *product*  $P \otimes Q$  is the interface automaton defined by:

$$\begin{aligned} S_{P \otimes Q} &= S_P \times S_Q \\ s_{P \otimes Q}^{\text{init}} &= s_P^{\text{init}} \times s_Q^{\text{init}} \\ \mathcal{A}_{P \otimes Q}^I &= (\mathcal{A}_P^I \cup \mathcal{A}_Q^I) \setminus \text{Comm}(P, Q) \\ \mathcal{A}_{P \otimes Q}^O &= \mathcal{A}_P^O \cup \mathcal{A}_Q^O \end{aligned}$$

and, for all  $\langle s, t \rangle \in S_P \times S_Q$ ,

$$\begin{aligned} \Gamma_{P \otimes Q}^I(\langle s, t \rangle) &= \left[ (\Gamma_P^I(s) \cup \Gamma_Q^I(t)) \cap \text{Prv}(P, Q) \right] \cup (\Gamma_P^I(s) \cap \Gamma_Q^I(t)), \\ \Gamma_{P \otimes Q}^O(\langle s, t \rangle) &= \left[ (\Gamma_P^O(s) \cup \Gamma_Q^O(t)) \cap \text{Prv}(P, Q) \right] \\ &\quad \cup \left[ (\Gamma_P(s) \cap \Gamma_Q(t)) \cap \text{Comm}(P, Q) \right], \end{aligned}$$

and, for all  $a \in \mathcal{A}_{P \otimes Q}$ ,

$$\delta_{P \otimes Q}(\langle s, t \rangle) = \begin{cases} \langle \delta_P(s, a), \delta_Q(t, a) \rangle & \text{if } a \in \mathcal{A}_P \cap \mathcal{A}_Q; \\ \langle \delta_P(s, a), t \rangle & \text{if } a \in \mathcal{A}_P \setminus \mathcal{A}_Q; \\ \langle s, \delta_Q(t, a) \rangle & \text{if } a \in \mathcal{A}_Q \setminus \mathcal{A}_P. \end{cases} \quad \blacksquare$$

As an example, the product of the interface automata *Ucomp* (Figure 4(a)) and *Netw* (Figure 4(b)) is depicted in Figure 4(c).

## 7.2 Composition

Since interface automata, differently from I/O automata, need not be input enabled, there may states in  $P \otimes Q$  where a communication action can be output by one of  $P$  or  $Q$ , but cannot be accepted as input by the other. These states correspond to *local incompatibilities* between  $P$  and  $Q$ .

**Definition 10 (locally incompatible states).** The set  $\text{Incmp}(P, Q)$  of *locally incompatible states* of two interface automata  $P$  and  $Q$  consists of all pairs of states  $\langle s, t \rangle \in S_P \times S_Q$  for which one of the following two conditions holds:

1. there is  $a \in \Gamma_P^O(s) \cap \text{Comm}(P, Q)$  such that  $a \notin \Gamma_Q^I(t)$ ,
2. there is  $a \in \Gamma_Q^O(t) \cap \text{Comm}(P, Q)$  such that  $a \notin \Gamma_P^I(s)$ .  $\blacksquare$

In the product between *Ucomp* and *Netw*, there is one state that corresponds to a local incompatibility; the state is depicted as shaded in Figure 4(c).

If  $P \otimes Q$  is closed then the presence of locally incompatible states indicates that  $P$  and  $Q$  are not compatible: in fact,  $P \otimes Q$  would be able to reach a locally incompatible state regardless of the environment. If  $P \otimes Q$  is open, on the other hand, there might be an input strategy that avoids all local incompatibilities. The states from which there environment can prevent reaching  $\text{Incmp}(P, Q)$  are called *usable*, emphasizing the fact that there is some way (some environment) to use them without giving rise to incompatibilities.

**Definition 11 (usable states).** A state  $u \in S_{P \otimes Q}$  is *usable* in  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$  if there is  $\pi^I \in \Pi_{P \otimes Q}^I$  such that, for all  $\pi^O \in \Pi_{P \otimes Q}^O$ , all  $\sigma \in \text{Outcomes}_{P \otimes Q}(u, \pi^I, \pi^O)$ , and all  $w \in \text{Incmp}(P, Q)$ , we have  $w \notin \sigma$ .  $\blacksquare$

In our case, the best input strategy to avoid locally incompatible states consists in restricting the sets of input actions to empty sets, since this minimizes the set of states that can be reached. Hence, we can give the following alternative characterization of the set of usable states.

**Theorem 2.** A state  $u \in S_{P \otimes Q}$  is usable in  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$  iff there is no path  $u_0, u_1, \dots, u_n \in S_{P \otimes Q}^*$  with  $u_0 = u$ ,  $u_n \in \text{Incmp}(P, Q)$ , and such that for all  $0 \leq k < n$ , there is  $a_k \in \Gamma_{P \otimes Q}^O(u_k)$  with  $\delta(u_k, a_k) = u_{k+1}$ .

We remark that this simple characterization of usable states — as the states that cannot reach local incompatibilities if no input actions are received — is a peculiarity of our choice of models for interfaces, rather than a general characteristic of interface theories. The simple characterization is due essentially to the weak expressive power of interface automata, and precisely, to the fact that in the absence of fairness, interface automata can express only what the environment *can* do (which input actions it can generate), but not what it *must* do. Consequently, the best input strategy consists in doing nothing.

The composition  $P \parallel Q$  is obtained by restricting  $P \otimes Q$  to its *usably reachable* states, that is, to the states that can be reached from the initial state under an input strategy that avoids all locally incompatible states.

**Definition 12 (usably reachable states).** A state  $u \in S_{P \otimes Q}$  is *usably reachable* in  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$  if there is  $\pi^I \in \Pi_{P \otimes Q}^I$  such that:

- for all initial states  $v \in s_{P \otimes Q}^{\text{init}}$ , all output strategies  $\pi^O \in \Pi_{P \otimes Q}^O$ , all outcomes  $\sigma \in \text{Outcomes}_{P \otimes Q}(v, \pi^I, \pi^O)$ , and all  $w \in \text{Incmp}(P, Q)$ , we have  $w \notin \sigma$ ;
- there is an initial state  $v \in s_{P \otimes Q}^{\text{init}}$ , an output strategy  $\pi^O \in \Pi_{P \otimes Q}^O$ , and an outcome  $\sigma \in \text{Outcomes}_{P \otimes Q}(v, \pi^I, \pi^O)$  such that  $u \in \sigma$ . ■

Alternatively, usably reachable states can be defined as the states that are reachable from the initial state of  $P \otimes Q$  by visiting only usable states.

**Theorem 3.** A state  $u$  of  $P \otimes Q$  is usably reachable in  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$  iff there is a sequence of states  $u_0, u_1, \dots, u_n \in S_{P \otimes Q}^+$  with  $u_0 \in s_{P \otimes Q}^{\text{init}}$  and  $u_n = u$ , and such that, for  $0 \leq k \leq n$ , we have:

1. the state  $u_k$  is usable in  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$ ;
2. there is  $a_k \in \Gamma_{P \otimes Q}(u_k)$  such that  $\delta(u_k, a_k) = u_{k+1}$ .

The composition  $P \parallel Q$  is obtained from the product  $P \otimes Q$  by restricting the latter to its usably reachable states.

**Definition 13 (composition).** Given two composable interface automata  $P$  and  $Q$ , let  $T$  be the set of usably reachable states of the product  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$ . The *composition*  $P \parallel Q$  of  $P$  and  $Q$  is the interface automaton defined by:

$$\begin{aligned} S_{P \parallel Q} &= T \\ s_{P \parallel Q}^{\text{init}} &= s_{P \otimes Q}^{\text{init}} \cap T \\ \mathcal{A}_{P \parallel Q}^I &= \mathcal{A}_{P \otimes Q}^I \\ \mathcal{A}_{P \parallel Q}^O &= \mathcal{A}_{P \otimes Q}^O \end{aligned}$$

and, for all  $u \in T$ ,

$$\begin{aligned} \Gamma_{P \parallel Q}^I(u) &= \{a \in \Gamma_{P \otimes Q}^I(u) \mid \delta_{P \otimes Q}(u, a) \in T\} \\ \Gamma_{P \parallel Q}^O(u) &= \Gamma_{P \otimes Q}^O(u) \end{aligned}$$

and, for all  $a \in I_{P\parallel Q}(u)$ ,

$$\delta_{P\parallel Q}(u, a) = \begin{cases} \delta_{P\otimes Q}(u, a) & \text{if } \delta_{P\otimes Q}(u, a) \in T; \\ \text{arbitrary} & \text{otherwise.} \quad \blacksquare \end{cases}$$

The composition between *Ucomp* and *Netw* (Figures 4(a) and 4(b)) is depicted in Figure 4(d).

### 7.3 Compatibility

We say that  $P$  and  $Q$  are *compatible* if their composition is non-empty.

**Definition 14 (compatibility).** Two composable interface automata  $P$  and  $Q$  are *compatible* iff  $s_{P\parallel Q}^{init} \neq \emptyset$ .  $\blacksquare$

The following alternative characterization of compatibility can be used to achieve efficient compatibility checking algorithms, in view of Theorem 2.

**Theorem 4.** *Two composable interface automata  $P$  and  $Q$  are compatible iff they are non-empty and  $\langle s, t \rangle$ , where  $s \in s_P^{init}$  and  $t \in s_Q^{init}$ , is usable in  $P \otimes Q$  with respect to  $\text{Incmp}(P, Q)$ .*

We now make precise the observation that the composition  $P\parallel Q$  consists exactly of the states of the product  $P \otimes Q$  that are reachable when  $P \otimes Q$  is in an environment that avoids all local incompatibilities  $\text{Incmp}(P, Q)$ . First, we define a *proper environment* to be an interface automaton  $E$  that closes  $P \otimes Q$ , and that prevents all local incompatibilities from being reached.

**Definition 15 (proper environment).** Let  $P$  and  $Q$  be two composable interface automata. A *proper environment* for  $P$  and  $Q$  is an interface automaton  $E$  such that the following conditions hold:

1.  $E$  is non-empty:  $s_E^{init} \neq \emptyset$ ;
2.  $E$  is composable with  $P \otimes Q$ ;
3.  $(P \otimes Q) \otimes E$  is closed;
4. for all  $u \in \text{Incmp}(P, Q)$  and all  $v \in S_E$ , the state  $\langle u, v \rangle$  is not reachable in  $(P \otimes Q) \otimes E$ .  $\blacksquare$

The composition  $P\parallel Q$  consists of the states of  $P \otimes Q$  that are reachable under a proper environment, justifying our Definitions 11, 12, and 13.

**Theorem 5.** *Let  $P$  and  $Q$  be two composable interface automata. For all  $u \in S_{P\otimes Q}$ , we have  $u \in S_{P\parallel Q}$  iff there is a proper environment  $E$  for  $P$  and  $Q$  and a state  $v \in S_E$  such that  $\langle u, v \rangle$  is reachable in  $(P \otimes Q) \otimes E$ .*

The following immediate corollary justifies, in retrospect, our definition of compatibility.

**Corollary 2.** *Two composable interface automata  $P$  and  $Q$  are compatible iff there is a proper environment for  $P$  and  $Q$ .*

## 7.4 Properties of Composition

The following theorem states that composition of interface automata is commutative and associative, modulo bi-equivalence.

**Theorem 6 (commutativity and associativity of composition).** *For all interface automata  $P$ ,  $Q$ , and  $R$ , if  $P$  and  $Q$  are composable, and if  $R$  is composable with  $P \otimes R$ , the following relations hold:*

$$P \parallel Q \simeq Q \parallel P, \quad (P \parallel Q) \parallel R \simeq P \parallel (Q \parallel R).$$

The following theorem states that refinement is compositional: if we refine one components of a composite system, we obtain a refinement of the global system.

**Theorem 7 (compositionality of refinement).** *For all interface automata  $P$ ,  $Q$ , and  $R$ , we have that if  $P \preceq Q$ , and if  $Q$  and  $R$  are composable, then also  $P$  and  $R$  are composable, and  $P \parallel R \preceq Q \parallel R$ .*

As a simple corollary of this theorem, we have that compatibility is preserved when we refine components in a design.

**Corollary 3 (refinement preserves compatibility, or substitutivity of refinement).** *For all interface automata  $P$ ,  $Q$ , and  $R$ , if  $Q$  and  $R$  are compatible, and if  $P \preceq Q$ , then also  $P$  and  $R$  are compatible.*

The relevance of this theorem lies in the fact that it justifies the use of interface automata as a formalism for specifying how components should interact in a design. Once a high-level design is drafted, consisting of the compatible interface automata  $Q_1, \dots, Q_n$ , we can refine each automaton  $Q_i$  into an implementation  $P_i$ , for  $1 \leq i \leq n$ , and obtain an implemented system  $P_1 \parallel \dots \parallel P_n$  in which all components are again compatible. Hence, the abstract interface automata  $Q_1, \dots, Q_n$  provide a specification that, if followed, guarantees compatibility of the implementations in a design [15].

## References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 1–17. Springer-Verlag, 1989.
2. S. Abramsky. Semantics of interaction. In *Trees in Algebra and Programming – CAAP'96, Proc. 21st Int. Coll., Linköping*, volume 1059 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1996.
3. S. Abramsky. Games in the semantics of programming languages. In *Proc. of the 11th Amsterdam Colloquium*, pages 1–6. ILLC, Dept. of Philosophy, University of Amsterdam, 1997.
4. S. Abramsky, S. Gay, and R. Nagarajan. A type-theoretic approach to deadlock-freedom of asynchronous systems. In *TACS'97: Theoretical Aspects of Computer Software. Third International Symposium*, 1997.

5. S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proc. 13th IEEE Symp. Logic in Comp. Sci.*, pages 334–344. IEEE Computer Society Press, 1998.
6. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.
7. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
8. R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR 98: Concurrency Theory. 9th Int. Conf.*, volume 1466 of *Lect. Notes in Comp. Sci.*, pages 163–178. Springer-Verlag, 1998.
9. J. Baeten and W. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
10. A. Chakrabarti, L. de Alfaro, T. Henzinger, and F. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Proc. of 14th Conf. on Computer Aided Verification*, volume 2404 of *Lect. Notes in Comp. Sci.*, pages 414–427. Springer-Verlag, 2002.
11. A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *Proceedings of the Third International Workshop on Embedded Software (EMSOFT 2003)*, *Lect. Notes in Comp. Sci.* Springer-Verlag, 2003.
12. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. 19th ACM Symp. Princ. of Prog. Lang.*, pages 343–354, 1992.
13. E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV 96: Proc. of 8th Conf. on Computer Aided Verification*, *Lect. Notes in Comp. Sci.*, pages 419–422. Springer-Verlag, 1996.
14. L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM Press, 2001.
15. L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT 01: 1st Intl. Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 148–165. Springer-Verlag, 2001.
16. L. de Alfaro, T. Henzinger, and F. Mang. Detecting errors before reaching them. In *CAV 00: Proc. of 12th Conf. on Computer Aided Verification*, volume 1855 of *Lect. Notes in Comp. Sci.*, pages 186–201. Springer-Verlag, 2000.
17. L. de Alfaro, T. Henzinger, and M. Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT 2002)*, volume 2491 of *Lect. Notes in Comp. Sci.*, pages 108–122. Springer-Verlag, 2002.
18. D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.
19. T. Henzinger and R. Alur. Alternating-time temporal logic. *J. ACM*, 49:672–713, 2002.
20. T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: methodology and case studies. In *CAV 98: Computer-aided Verification*, volume 1427 of *Lect. Notes in Comp. Sci.*, pages 440–451. Springer-Verlag, 1998.
21. T. Henzinger, S. Qadeer, S. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-aided Design*, *Lecture Notes in Computer Science* 1522, pages 421–432. Springer-Verlag, 1998.
22. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
23. R. Keller. Formal verification of parallel programs. *Comm. ACM*, 19(7):371–384, 1976.

24. O. Kupferman and M. Vardi. Module checking. In *CAV 96: Proc. of 8th Conf. on Computer Aided Verification*, volume 1102 of *Lect. Notes in Comp. Sci.*, pages 75–86. Springer-Verlag, 1996.
25. N. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
26. N. Lynch and M. Tuttle. Hierarcical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. Princ. of Dist. Comp.*, pages 137–151, 1987.
27. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
28. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.
29. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1980.
30. R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 1202–1242. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.
31. J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
32. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190. ACM Press, 1989.
33. A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 652–671, 1989.
34. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of 12th Annual Symp. on Theor. Asp. of Comp. Sci.*, volume 900 of *Lect. Notes in Comp. Sci.*, pages 1–13. Springer-Verlag, 1995.